KE 1

Einleitung

Datenbank Management System (DBMS)

konsistente, redundanzarme alleinige Repräsentation miteinander verbundener Datensätze **(DB)** durch Software mit Datenmodell, Abfrage- und Manipulationssprache gemanaged **(DBMS)** => Datenunanhängigkeit, Transaktionen, Integritätsbedingungen und Zugriffsrechte.

Klassische DB

einfache Anwendungen und *einfache Datentypen*. **Beschränkungen** bei:

- geometrischen Formen (außer Punkte) => **Raum**
- zeitliche Entwicklungen (nur Gegenwart, nicht Vergangenheit) => **Zeit**
- bewegliche Objekte (nur durch nicht praktikable ständige Updates) => **Bewegung**

Spatial DB (Raum)

Erweitern DBMS Datenmodelle und Abfragesprachen für Geometrien:

- **Spatial Algebra** (zB ROSE Algebra): Kollektion von **Spatial Data Types (SDT)** -> kapseln Objektabstraktionen (Punkt, Linie, Region) = **Datenstrukturen** mit Operationen = **Algorithmen**.
- **Indexierungstechniken** für mehrdimensionalen Raum, zB **R-Baum**, in dem SDTs durch MBBs repräsentiert werden.
- Erweiterungen des **Optimierers** (Übersetzung, Kosten)

Modellierung räumlicher Konzepte

Entitäten: alles was sich auf Landkarte findet

- **Objekte:** Punkt, Linie, Region
- **Kollektionen:** (verschachtelte) Partition, Netzwerk

Temporal DB (Zeit)

Konzept

Standard-DB verwalten Gegenwart. **Update** bewirkt Zustandswechsel, vorheriger

Zustand geht verloren. Geschichte nur über **explizite Zeitattribute** (Datentypen date und time). Ziel temporal DB: Zeitkonzepte tief in DBMS- Datenmodell und -Abfragesprache integrieren.

- **Zeitdomäne:** 1D (Vergangenheit -> Zukunft), begrenzt|unendlich, diskret|stetig, absolut|relativ
- **Datentypen:** instant, period, intervall, date, time, timestamp
- **Zeitdimensionen (Semantik):** valid | transaction time
- **temporal DB:** snapshot | valid-time/historical | transaction-time/rollback | bitemporal

Erweiterungen des Datenmodells

- **facts:** Elemente Datenmodell des DBMS
- **timestamps:** Elemente Zeitdomäne

Relationale Modelle:

	nur valid-time	*bi-temporal*
tupel-timstamping	**SEGEV** instant timestamps **SARDA** period timestamps	**BCDM (Bitemporal Conceptual Data Model)** Aufzählung aller bitemporaler chronons eines Tupel - BTEs
attribute-timestamping	**HRDM (Historical Relational Data Model)** Attributwerte als Funktionen Zeit -> Domäne	**BHARGAVA** timestamp als Rechteck

Erweiterungen der Anfragesprache: TSQL 2 (basiert auf BCDM)

6 Arten von **Relationen**:

1. snapshot
2. as valid (state | event)
3. as transaction
4. as valid (state | event) and transaction

state = facts true over period, **event** = at instant of time
coalescing: mergen von Tupelinstanzen durch Vereinigung ihrer valid-time Perioden.

Moving Objects (Bewegung)

Ziel: jegliche Art bewegliche Objekte in DB darstellen können. **2 Perspektiven:**

Location Management Perspektive

Verwalten der Positionen einer Menge von beweglicher Entitäten in einer DB. *Problem:*

Viele Updates -> hohe Kosten, wenige Update -> ungenaue Positionen. *Lösung:* Position als Funktion der Zeit = **Motion Vector** (Position mit Geschwindigkeit + Richtung). Für Gegenwart und nahe Zukunft.

Spatio-Temporal Data Perspektive

Nicht nur gegenwärtigen Zustand sondern gesamte **History of development** abspeichern. Zustand, Veränderungen, Beziehungen abfragen und analysieren.

- **Daten:** geometrische Objekte und Kollektionen von Objekten
- **Veränderungen**: diskret | stetig

Klassifikation ST-Data in 3D (2D Raum + 1D Zeit):
Zeit (instant|period) x Raum (point|region)

Ansätze:
- **temporal-DB** um **räumliche Datentypen** erweitern -> natürlich, aber keine moving objects
- **ST-Datentypen** -> abstrakte ST-Datentypen (mpoint, mregion) mit geeigneten Operationen (trajectory, intersection...)

Spatio-Temporal Databases in der Vergangenheit

Spatio-Bitemporal Objects

Vereinheitlichter Ansatz für räumliche und zeitliche Information, basierend auf Konzepten rein **räumlicher Modellierung** (*2D Simplicial Complexes)* und rein **zeitlicher Modellierung** (orthoginale Dimensionen *Transaction- und valid Time*). Als *ADT* modelliert und als *Attributtyp* in DBMS eingesetzt.

Bitemporale Elemente (BTE)

Vereinigung endlicher Menge disjunkter kartesischer Produkte von **Transaction-Time-** und **Valid-Time-Intervallen** => Punktmenge in der Ebene. *Semantik:* Zur Zeit T enthält DB Information das Object zur Zeit V existierte

Spatial Objects als Simplicial Complexes

Basiert auf **algebraischer Topologie:** topologische Strukturen zur Klassifizierung und formalen Beschreibung von Punktmengen mit algebraischen Mitteln (**Problemübersetzung**: Knoten und Kanten statt Punkte und Linien).

Spatial Object:
- **Simplex:** in Dimension k , bestehend aus k+1 unabhängigen Simplexes der Dimension k-1.
- **Simplicial Complex:** endliche Menge von Simplexen mit **completeness of incidence**

(keine 2 Simplexe in C beanspruchen [teilweise] denselben Platz, Linien schneiden sich nur an Start und Endpunkten) und **completeness of inclusion (**keine geometrischen Anomalien).

Spatio-Bitemporal Objects:
BTEs als *Label* an Komponenten von **simplicial complexes** anheften.

- **ST-Simplex:** geordnetes Paar (Simplex, BTE)
- **ST-Complex:** endliche Menge ST-Simplexe, für die gilt => nur 1 BTE pro simplex, räumliche Projektion ergibt simplicial complex, zeitliche Projektion einer Fläche eines ST-simplex muß in zeitlicher Projektion ST-Simplex enthalten sein (simplex kann nicht ohne seine Flächen existieren)

Spatio-Bitemporal Operations:
zb equal, subset, boundary, (S|T)-Project, union, intersection...

Event-basierter Ansatz

Position in der Zeit ist die primäre organisatorische Basis um Veränderungen zu speichern. **Ereignis:** Zustandswechsel zu einem bestimmten Augenblick. Nur **valid-time**! Zu t_0 *Basemap* speichern. Bei t_1, t_2 ... alle **Veränderungen** (nur die *deltas)* als **Funktion diskreter Zeit in die Domäne** speichern.
Verbesserungen:
- **value specific groups (=component):** alle Positionen mit dem gleichen shared value
- **raster run length encoding:** innerhalb jeder Komponente

KE 2

Modelling and Querying current Movement

Location management

In DBMS sind **motion vectors** nicht direkt sichtbar: dienen als Implementierung für die abstraktere Sicht der **dynamischen Attribute** => verändern ihre Werte im Zeitablauf ohne explizite Updates (enthalten Information über die *Gegenwart* und unsichere Information über die *nahe Zukunft)*. **Datentyp** (zB point) und **Query** bleiben unverändert – nur Abfrageergebnisse ändern sich mit der Zeit. **Vertrag** zwischen *moving object* und *DB*: wenn **deviation > Grenzwert** -> Update mit aktueller Position, Geschwindigkeit und evtl neuer Grenzwert.

MOST (Moving Objects Spatio Temporal Data Model)

Konzept

Datenmodell für **gegenwärtige und zukünftige Bewegung.** Annahmen:

- **DB:** Menge von Objektklassen (gegeben durch ihre Attributmengen)
- **Spatial data types mit Operationen:** point, line, polygon
- **Spatial Object class:** point class, line class, polygon class
- **Spezielles Object Time** -> vergrößert sich bei jedem clock-tick

Dynamische Attribute (neue Idee in MOST)

Attribute einer Objektklasse entweder **statisch** oder **dynamisch**. **Dynamisches Attribut** formal durch **3 Sub-Attribute** repräsentiert: **A.(value| updatetime | function)**.
Sematik: [dynamic] value (A,t) = A.value + A.function(t – A.updatetime)

Repräsentation von Objekt-Positionen

- **Positionsattribut pos:** freie Bewegung in (x,y)Ebene. **Uncertainty:** around Position (x,y)
- **Attribute loc.(route|startlocation|starttime|direction|speed|uncertainty):** auf Netzwerk. Von startlocation bewegt sich Object mit speed auf route und befindet sich zu starttime + t auf route in Distanz speed*t von startlocation in Richtung direction. **Uncertainty:** vor/hinter pPosition (x,y).

DB-Zustand

Abbildung die mit jeder **Objektklasse** eine Menge von Objekten des entsprechenden Typs und mit dem T**ime-Objekt** einen Zeitwert verbindet. Jeder *clock-tick* -> neuer DB-Zustand.

DB-Histories

unendliche **Folge von DB Zuständen** (*past history* < t, *future history* > t). Jedes Update -> neue DB-History.

Drei Arten von Queries

Queries sind Prädikate über DB-Histories (anstatt über einzelne Zustände). Diesselbe **Anfrage** kann in **3 Modi** gestellt werden:

- **instantaneous query: $Q(H_t, t)$** -> evaluiert auf mit t (=current time) beginnender History
- **continuous query: $Q(H_t, t)$, $Q(H_{t+1}, t+1)$, ...** -> evaluiert als neue instantaneous query auf jedem clock-tick
- **persistent query: $Q(H_{t,0}, t)$, $Q(H_{t,1}, t)$, ...** -> nur für temporale Version MOST-Model, da Information über Vergangenheit nötig. Ergebnis ändert sich bei expliziten Updates.

FTL – eine auf Future Temporal Logic basierende Abfragesprache

Konzept

Erlaubt Bedingungen über die Zukunft auszudrücken:
RETRIEVE <target-list> FROM <object-classes> WHERE <FTL-Formel>
FTL-Formel: aus Termen werden well-formed formulas gebildet => **syntaktische Primitive**, aus denen wieder **abgeleitete Notationen** wie eventually, always, until_within_c ... gebildet werden.
Nur für **instantaneous queries** (continuous query muß ausserhalb spezifiziert werden).
Folgen von Bedingungen sowie **Grenzen für Zeitperioden** sind möglich.
MOST Modell und FTL Sprache auf **DBMS** mit eigener **non-temporaler Abfragesprache** aufgesetzt. FTL erlaubt **eingebettete Queries** in dieser Sprache.

Evaluierung FTL-Queries:

FTL-Formel wird in **Subformeln** strukturiert. Zuerst Relationen für **atomare Formeln** berechnet, dann Formeln von unten nach oben evaluiert. Eine Menge von **Tupeln T** mit derselben **Instanzierung ρ** und einer Menge von (disjunkten) **Zeitintervallen *int*** repräsentiert eine Kombination von Objekten die die FTL-Formel während *int* erfüllen.

Location updates – Balancieren von Update Kosten und Imprecision

Position eines moving objects in DB ist naturgemäß unpräzise. Verschiedene Update policies, zB **dead-reckoning policies**: updates wenn Abweichung > Grenzwert.

- **Speed-dead-reckoning (sdr) policy:** konstanter Grenzwert während gesamter Reise
- **Adaptive-dead-reckoning (adr) policy:** bei jedem Update neuer Grenzwert (auf Basis Durchschnittskosten) => totale Informationskosten minimiert.
- **Disconnection-detection-dead-reckoning (dtdr) policy:** fraktional abnehmender Grenzwert

Konzepte von **Ungenauigkeit (Imprecision)**

- **Abweichung (deviation):** Distanz DB-Position <-> aktuelle Position
- **Unsicherheit (uncertainty):** Größe der Fläche die alle möglichen Positionen des mObjects einschließt

=> Erzeugen Kosten proportional zu ihrer Größe. Weiterer Kostenfaktor:
communication/transmission cost: updates von GPS über kabelloses Netzwerk zu DB.
Informationskosten einer Reise als Summe der 3 Kostenarten über die betreffende Periode.

Uncertainty der trajectory von moving objects

Trajectory: sowohl *motion plan* für die Zukunft als auch *history of movement*. Als Polyline bzw als zylindrischer Körper (uncertainty aspect berücksichtigt) im 3D (2D+1D) Raum modelliert.

- Possible **motion curve** innerhalb **trajectory volume**
- Possible **route** innerhalb **uncertainty zone** (2D Projektion)

Querying:
- **point queries:** 2 Operatoren, die sich auf einzelne trajectory beziehen (whereAt, whenAt)
- **spatio-temporal queries:** 6 boolsche Prädikate, die erfüllt sind, wenn mObject sich innerhalb Zeitintervall in Region befindet -> für ST-range queries geeignet. Existenz-|All-Quantifikation bzgl **Zeit** (sometimes|always), **Raum** (somewhere|[everywhere*]) und **Unsicherheit** (possibly|definitely) => 8 Operatoren *//* nicht sinnvoll*

Algorithmen für Spatio-Temporal Operationen und Prädikate

point queries

- whereAt -> binäre Suche und lineare Interpolation
- whenAt -> jedes Segment Trajectory in konstanter Zeit examiniert

spatio-temporal predicates

Idee: 3D-Problem auf 2D-Problem reduzieren und leichter lösen.
Jedes Segment trajectory volume durch MBB angenähert (als Filter und 3D Indexing Schema). Query polygon r + Zeitintervall -> Query Prisma.

- **Filtering step:** MBB schneidet query prism -> *Candidate set*
- **Refinement step:** exakter Algorithmus auf Segmentes der Kandidatenmenge

Minkowski Summme: alle Elemente eines Polygons q, seiner Umrandung sowie und eines Sweeps einer Kreisscheibe d(c) mit Radius c entlang der Umrandung.

Beispiele für **Algorithmen:**
- **possiblySometimes:** wenn Schnittmenge(SpatialProjection(trajectory), MinkowskiSum(r , d(loc.uncertainty))) != leere Menge => FALSE, sonst TRUE
- **possiblyAlways:** wenn SpatialProjection(trajectory) komplett in MinkowskiSum(r , d(loc.uncertainty)) => TRUE, sonst FALSE

Modelling and Querying History of Movement

Konzept

Datemodell und **Abfragesprache** für **History of Movement** (anstatt Bewegung in Gegenwart und naher Zukunft). Design von **Spatio-Temporal Datentypen** als **abstraktes Modell** und Übersetzung in **diskretes (implementierbares) Modell**. Komposition von Prädikaten um **Developments** der Konfigurationen von moving objects zu beschreiben.

Ein auf abstrakten Datentypen basierender Ansatz

Typen und Operationen

Zeitabhängige statial entities (mpoint, mregion) als **ADT** mit **passenden Operationen** repräsentiert -> als **Attributtypen** in DBMS-Datenmodell (relational|OO) eingebettet (Interaktion mit DBMS-Query Language).

- **Non-temporal types**: base (int, bool) , spatial (line, region), time (instant, period)
- **temporal types (partielle Funktionen Zeit -> Domäne):** stetig (mpoint,mregion, mreal), diskret (mint, mstring, mbool)

Untersuchung von Operationen führt zu Bedarf nach neuen Datentypen.

Design einer many-sorted-Algebra

System von zusammengehörigen Typen und Operationen das erlaubt alle interessanten Fragen zu stellen und in sich geschlossen ist.

1. **Signatur** mit **sorts** (= Namen für Typen) und **Operations** definieren, also *Operationen* mit ihren *Argument-* und *Ergebnistypen*.
2. **Semantik der Signatur** definieren, also **carrier sets** (Trägermengen) für die *sorts* und **functions** für die *Operationen*

Abstraktes vs. Diskretes Modell

3 Begriffspaare: abstrakt <-> konkret, stetig <-> diskret, unendlich <-> endlich

- **Abstrakte Ebene:** Definitionen auf Basis unendlicher Mengen. Konzeptionell einfacher und einheitlicher (generisch). Bewegung von Real-World Objekten ist stetige Funktion der Zeit. Nicht implementierbar.
- **Diskrete Ebene:** ein oder mehere diskrete Modelle (-> nur endliche

Repräsentationen) als Instanzierungen des abstrakten Modells implementieren.

Das abstrakte Modell

Anforderungen

System ST-Datentypen plus Operationen für Repräsentation und Abfrage vom mObjects: **closure** | **genericity** | **consistency** non-temporal (types|operations) <-> temporal (types|operations).

Datentypen

- **Basistypen** *a* aus {int, real, bool, string}
- **Spatial types** *s* aus {point, points, line, region}
- **Time type** -> instant *inst*
- **Temporal types** -> moving(*a* | *s*)
- **Range types** -> range(*a)* bzw range(s) = (points|line|region) bzw period = range (*inst*)

Formale Definition Datentypen

Für jeden Datentyp muß eine Menge möglicher Werte, die **Trägermenge (Carrier Set)** definiert werden. Zuerst **Struktur** (C- oder R-Complex und assoziierte Punkmenge) definieren und dann eine Linie/Region als **Punktmenge** definieren, die zu einem solchen Complex gehören könnte. zB

- **line:** Untermenge von R^2 -> unendliche Menge von Punkten. Als **C-complex** => endliche Menge von *einfachen Kurven* in *kanonischer Reräsentation* (entsprechen Kanten eines Graphen)
- **region:** als **R-Complex** => endliche Menge **regular closed sets** (Q=closure (interior)), Grenzline als *C-Complex*

Operationen

Design-Prinzipien:

- so **generisch** wie möglich: Jeden Datentyp entweder mit **1D** oder mit **2D Raum** verbinden und dann innerhalb jeden Raumes zwischen Typen unterscheiden, deren Werte einzelne Elemente des Raumes sind **(point types)** und Typen deren Werte Untermengen des Raumes sind (**point set types).** Definition der generischen Operationen dann in Bezug auf Räume und ihre verfügbaren point und point set types.
- **Konsistenz:** Operationen auf nontemporal types entwerfen und systematisch und einheitlich auf temporal types ausdehnen => **lifting:** jeder der Argumenttypen einer Signatur kann zu zugehörigen temporalen Typ ersetzt werden.
- die **interessanten Phänomene** einfangen: Operationen aus verschiedenen Zweigen der Mathematik (Mengenlehre, first order logic, topology...)

Klassifikation **non-temporal types:**
[1D (diskret|stetig) | 2D] x [point type | point set type]

Klassifikation **Operationen** auf **non-temporal types:**
Prädikate | Mengenoperationen | Aggregation | Numerische Eigenschaften von Mengen | Entfernung und Richtung |

Klassifikation **Operationen** auf **temporal types:**
alle moving(a) - Typen sind partielle Funktionen Zeit -> Domäne

- **Projektion** auf Domäne und Range
- **Interaktion** mit Domäne und Range
- **Rate of Change** (Ableitung)

Operationen auf **Mengen von Objekten:** mit **decompose** Elemente von point set types in Queries zugreifbar machen.

Das diskrete Modell

Konzept

Es existieren viele Möglichkeiten die unendlichen Punktmengen des abstrakten Modells zu implementieren (verschiedene Instanzierungen).

- **Non-temporal types:** Programmiersprachliche Primitive oder lineare Approximation als polyline oder polygon
- **Temporal types:** sliced representation von moving objects. **Unit types:** Paar bestehend aus Zeitintervall und Wert (der einfache Funktion beschreibt). **Unit** ist Funktion der Zeit während Zeitintervall. **Sliced representation** als Ganzes durch **mapping-Typkonstruktor** gemanaged => Menge von Units mit paarweise disjunkten Zeitintervallen.

KE 4

Modelling and Querying History of Movement

Spatio-temporal predicates and developments

Konzept

Prädikate sind in DB-Systemen wichtig als **Filterbedingungen** in **selection** und **join**. ST-Prädikate und -Developments ermöglichen **ST-selection** und **ST-join** zwischen zwei mPoints, zwei mRegions oder mPoint/mRegion. Veränderungen in der Zeit führen zu Modifikationen der **topologischen Beziehungen** zwischen mObjects.
Development: Sequenz räumlicher Beziehungen, die zu bestimmten Zeitpunkten (-intervallen) gültig sind. Alternierende Sequenzen von S- und ST-Predicates. Können normalisiert werden.

Topologische Prädikate für spatial objects

9-Intersection -model: 9 mögliche Überschneidungen *boundary*, *interior* und *exterior* A und B. Prüfung bzgl **topologische Invarianzkriterien „leer/nicht-leer“** -> 29 = 512 Konfigurationen, davon 8 sinnvoll: equal | disjoint | coveredBy | covers | overlap | meet | inside | contains.
Topologische Transitionen im **conceptual neighborhood graph** dargestellt: Knoten -> topologische Beziehungen, Kanten -> konzeptuelle Nachbarschaft. Nur konzeptionelle Nachbarn können direkt erreicht werden. **Topologische Distanz:** Anzahl unterschiedlicher TRUE, FALSE Einträge in 9-Intersection-Matrix.

Temporal lifting von topologischen Prädikaten

Die Konzepte für temporales Liften zeitunabhängiger Operationen können auf **topologische Prädikate** angewandt werden: aus

- *spatial x spatial -> bool* wird
- *spatio-temporal (= moving(a)) x spatio-temporal (=moving(b)) -> moving(bool)*

Temporal Aggregation

Ein geliftetes **spatial predicate** liefert definierten Wert nur auf der Schnittmenge der Zeitdomänen zweier ST-Objects. **Existenzielle** und **universale Aggregation** bestimmen Gültigkeit **ST- Prädikat** -> beide Objekte total definiert (instant) | gleiche Zeitdomäne (Vereinigung)| Projektion auf Zeitdomäne eines Objektes | Schnittmenge der Zeitdomänen. Beispiele:

- Disjoint := $\forall \cap disjoint$
- Meet := $\forall \cup meet$ etc.

Eine Algebra für Spatio-Temporale Prädikate

Objekte: ST-predicates
Operationen:

- **temporale Alternative:** P|Q (kann immer in development ohne Alternative normalisiert werden)
- **Prädikat Negation:** ~P
- **Prädikat Reflektion:** P<-

Kanonische Kollektion ST-Predicates:

- **point/point** development graph: 13 Pfade

- **point/region** development graph: 28 Pfade
- **region/region** development graph: 2198 Pfade

zuviele mögliche Prädikate -> kleine anwendungsspezifische Auswahl für User oder Sprache für eigene Prädikatkonstruktion.

Querying developments in STQL

Integration von ST-predicates und Querying von Developments mit SQL-ähnlicher **ST-Query -Sprache**. **Makros** ermöglichen die Definition **zusammengesetzter Prädikate**.

KE 5

Datenstrukturen und Algorithmen für Moving Object Types

Konzepte

Datenstrukturen für die Datentypen des diskreten Modells und **Algorithmen** für die Operationen.

- **Abstraktes Modell:** alle Operationen anwendbar auf alle Kombinationen von Argumenttypen -> überladene Operationen mit sehr großer Menge an Funktionalität
- **Implementierungsebene:** Anzahl Algorithmen steigt noch, da verschiedene Argumenttypen für eine überladene Operation nicht immer vom selben Algorithmus verarbeitet werden können.

Datenstrukturen

Allgemeine Anforderungen und Strategie

Werte von als Attributtypen in DB-Umgebung genutzten Datentypen werden unter Kontrolle DBMS im Speicher plaziert => keine Pointer, Repräsentationen aus kleiner Anzahl Speicherblöcken die effizient zwischen Haupt- und Sekundärspeicher verschoben werden können.

Datentypen generell durch **root record** dargestellt. Sollen Ansprüchen unterschiedlicher Algorithmen genügen (ganze Algebra im Blick). Bestehen aus:

- einige **Komponenten fester Größe**
- Referenzen auf Arrays (**Komponenten variabler Größe**). **Zeiger** als Array-Indices. **Mengenwertige Datentypen** als Arrays mit eindeutiger Ordnung -> erleichtert Vergleiche.

Non-temporal data types

- **base types:** record mit korrespondierendem **Programmiersprachenwert** und **boolean flag** (-> definiert?).
- **spatial data types:** benutzen alle den **Typ coordinate**. **Point:** record (x,y) + boolean flag. **Points:** root record -> Array mit Punkten. **Line:** root record -> Array von Halfsegments (für *plane sweep algorithmen)*. **Region:** root record -> 3 Arrays (segments, cycles, faces). Alle außer point: **summary fields** für Zugriff in O(1).
- **range data types:** root record mit Array, dessen Werte **intervall records** mit totaler Ordnung sind.
- **Intime value:** record(instant, value) mit value vom korrespondierenden Datentyp.

Temporal data types

werden alle durch **sliced representation** dargestellt => Wert eines temporal (=moving) data types als Menge von **units** = record(interval, unit function f: zeit -> non-temporal type).

- **diskrete Typen:** unit function ist Wert des korrespondierenden non-temporal types
- **stetige Typen: ureal** -> f: (Quadratwurzel) quadratisches Polynom, **upoint** -> f: lineare Funktion(x, y), **uregion** -> f: record mit 3 Arrays (msegments, cycles, faces)

Summary fields: auf object-level und auf unit-level

Algorithmen für Operationen auf Temporal Data Types

Allgemeine Betrachtungen

- **Refinement Partitions:** für viele Operationen gültiges algorithmisches Schema -> als **vorbereitender Schritt** für jede binäre Operation mit 2 moving arguments. **Zerlegen der units** in zeitlich kürzere units, so dass units beider Argumente entweder auf demselben oder auf zwei disjunkten Zeitintervallen definiert sind. Als **nachbereitenden Schritt** vieler Operationen -> **merge benachbarter units** auf den Ergebnistypen.
- **Filtering Approach:** jeder Algorithmus filtert seine Argumente aufgrund von **Hilfsinformationen** (zB summary fields). Erst **filter-step** (auf MBR), dann **refinement-step** (auf Datentyp).

Projektion auf domain und range

Operationen, die **moving** oder **intime Wert** als **Operand** nehmen und verschiedene Arten von **Projektionen** auf die **Zeitkomponente (domain)** oder die **Funktionskomponente (range)** berechnen:
deftime | rangevalues | locations | trajectory | traversed | inst/val (von intime-typ)

Interaction mit domain und range

Beschränken **moving entity** auf auf **Instanz/Periode** der Zeitdomäne oder auf **Wert/Wertebereich** (aus range):
atinstant | atperiods| initial/final | present | at | atmin/atmax | passes

Rate of change

Wichtige **Eigenschaft** jedes **zeitabhängigen Wertes.** Globales **algorithmisches Schema: Scannen** der Abbildung der units von a und **berechnen** der zugehöriges Ergebnisunits unter **Verschmelzung benachbarter Ergebnisunits** mit demselben Wert.
Derivative | derivable | speed | velocity

Algorithmen für geliftete Operationen

Ursprünglich für non-temporal objects definierte Operationen.

- **Prädikate:** isempty | =/≠ | intersects | inside
- **Mengenoperationen:** regularisierte Mengensemantik. Intersection | union | minus
- **Aggregation: unlifted mode:** Punktmenge -> Punkte, **lifted mode:** Punktmenge -> Punkte in gesamter Lebensspanne von mObject.
- **Numerische Eigenschaften:** no_components | perimeter | area
- **Distanz und Richtung:** distance | direction
- **Boolean:** and | or | not

Constraint Database Approach (Part I)

Konzepte

Radikal unterschiedlicher Ansatz für die Modellierung und Manipulation von spatio/spatio-temporal values oder moving objects. Zwei Grundgedanken:

- spatial objects sind **Punktmengen im k-dimensionalen Raum** => Relation mit k Attributen, eines für jede Dimension. Relationen idR **unendlich.**
- **Einfache Formeln** (first order logic mit constraints) für **endliche Repräsentationen.**

Gleiche Idee wie beim abstrakten/diskreten Modell: **konzeptionelles Modell** um Typen auf unendlichen Mengen zu definieren und definiere **diskretes Modell** mit endlichen Repräsentationen der Typen. Hier nur ein einziger abstrakter Typ: **infinite relation,** sowie eine Klasse von diskreten Modellen: **constraint relations.**

Ein abstraktes Modell – Infinite Relationen

Zwei Ansätze:
- **Flat relations:** reguläre (unendliche) Relationen. Spatial types werden nicht mehr

benötigt, werden einfach durch **(x,y)-Koordinaten** repräsentiert.
- **Nested relation/ ADT:** Menge von spatial objects als endliche Menge von Formen, die jeweils unendliche Punktmengen darstellen. **Punktmengen** von spatial objects durch **geometrische Typen** in **Untermengen** gruppiert.

Viele **Queries** auf **spatial data/moving objects** können im konzeptionellen Modell unendlicher Relationen mit Standard relationaler Abfragesprache einfach und natürlich formuliert werden. **Hauptvorteile Modell:**

- **Benutzerfreundlich:** user muß nicht viele verschiedene Operationen und Datentypen lernen.
- Modell **dimensionsunabhängig** -> spatial data in beliebigen Dimensionen.

Ein diskretes Modell: Constraint Relationen

Idee: eine **Relation** als **logische Formel** über bestimmte **atomare constraints** beschreiben -> enthält genau die Punkte, die Formel wahr machen. **Formeln** mit **logischen Verknüpfungen** AND, OR, NOT sowie **Existenz- und Allquantoren** gebildet.

Darstellung einfacher **geometrischer Formen** mit **constraint-Formeln:**

- **Punkt:** Konjunktion zweier Gleichungen
- **Linie:** lineare Gleichung
- **Liniensegment:** lineare Gleichung in Konjunktion mit 2 Ungleichungen
- **Halbebene:** lineare Ungleichung
- **konvexes Polygon:** Konjunktion von Halbebenen.

Symbolischer Tupel: Konjunktion von constraints
Symbolische Relation: Disjunktion einer Menge von symbolischen Tupeln.

Klassisches relationales Modell ist Spezialfall des **Constraint-Modells:**
erlaubt nur **equality constraints (=)** mit einer einzelnene Variablen in der Form **Attribut = Value.**

Die **präzise Form** des **atomaren constraints** bestimmt, welche geometrischen Formen dargestellt werden können:

- **polynomiale constraints:** sehr expressiv, schwer zu implementieren
- **lineare constraints:** für lineare Formen wie Linien, Polygone, Hyperebenen und Polytope (=> korrespondieren mit bisher benutzten Repräsentationen von spatial types und moving objects).

Das lineare Constraint Datenmodell

Lineare constraints können als atomare Formeln in first order logic eingefügt werden.

Spezielle Normalform (<=> Definition symbolische Relation):

- Formel ist **frei von Quantoren: Allquantoren** in **extistenzielle Quantoren** transformieren und diese mit speziellem Algorithmus eliminieren.
- Formel ist in **disjunktiver Normalform (DNF):** $f_1 \cup f_2 \cup \ldots \cup f_n$ mit f_i = Konjunktion atomarer linearer constraints (<=> Definition symbolischer Tupel).

RelationaleAlgebra für Constraint – Relationen

Relationale Abfragesprachen wie **SQL** basieren auf **first order logic (relational calculus/algebra). Evaluierung:** Abfragesprache -> relationale Algebra -> spezielle Evaluierungstechniken. Wenn man es schafft eine **Version der relationalen Algebra** zu definieren, die **lineare constraint Relationen** manipuliert, nähert man sich der Implementierung einer **Abfragesprache für unendliche Relationen.**

7 Operationen:
union | intersection | minus | selection | projection | cartesian product | natural join

- **Selection:** constraint der Abfrage jedem symbolischen Tupel der symbolischen Relation hinzufügen. Inkonsistenzen und Redundanzen entfernen, dann evaluieren.
- **Intersection** <=> **cartesian product** <=> **join:** forme alle Paare von symbolischen Tupeln aus den beiden Argumentrelationen.
- **Projection:** mit **Fourir-Motzkin-Eliminationsmethode** -> Elimination einer Variablen y aus Konjunktion von Constraints C (<=> Projektion k-dim Polyeder in k-1 dim Raum). **Zwei Schritte: (1)** Isoliere y innerhalb jeden constraints -> 2 Untermengen von C: $y \geq a_i$, $y \leq b_i$. **(2)** ersetze C durch Menge von Constraints $\{a_i \leq b_i\}$.

KE 6

Constraint Database Approach (Part II)

Implementierung des Constraint Modells (DEDALE-Prototyp)

Repräsentation von Relationen

- **top-level Relationen (finite) -**> wie in Standard-DB Systemen als Menge von Records innerhalb Datei repräsentiert. Record und File Abstraktionen von Storage Manager Komponente zur Verfügung gestellt (sorgt für Transaktionen, Logging, Recovery etc.)

- **symbolische sub-Relationen oder constraint-formeln (infinite)** -> wie andere nicht-Standard Attribute als binäre Speicherobjekte variabler Größe gespeichert.

Repräsentation von symbolischen Relationen (Constraint Formeln)

Relationen in **Normalform**: DNF | keine redundanten/inkonsistenten Constraints | Constraints innerhalb Tupel geordnet (-> Konversion in Vector-Repräsentation auf UI).
Repräsentation:
- **linearer constraint -**> constraint byte block (kompakter Speicherblock: Sequenz von Bytes)
- **constraint tupel** -> tuple byte block (Sequenz von constraint byte blocks)
- **constraint relation** -> relation byte block (Sequenz von tuple byte blocks). Symbolische Relation als Paar (mbb, ref) gespeichert, mit ref: Zeiger auf relation byte block.

Data Loading and Conversion

Wie kann **spatial data in 2D** (points, polylines, polygones) aus dem Vectorformat in **Constraint Relationen in Normalform** konvertiert werden?
Grundlegende Anforderung: Dekomposition in konvexe Formen:
- **Point:** -> constraint tupel
- **Polyline:** -> symbolische Relation mit einem Tupel pro Liniensegment, bestehend aus 5 constraints (1 für Linie, 4 für mbb).
- **Polygon:** Kombination von einfachen Algorithmen für Zerlegung in nicht optimale, aber nicht zu große Anzahl von Komponenten. **3 Schritte: (1)** Zerlegung Polygon in (leicht zu triangulierende) **monotone Polygone, (2) Triangulation** der monotonen Polygone und **(3)** Bilden **konvexer Polygone** aus der Triangulation.

Normalisation symbolischer Tupel

Eine Konjunktion von Constraints in Normalform transformieren: Erfüllbarkeit prüfen und redundante Constraints entfernen. Constraints scannen und **Anzahl Gleichheitsoperatoren ("=")** feststellen ->
- **> 2** -> Tupel nicht erfüllbar
- **= 2** -> Punkt
- **= 1** -> Liniensegment
- **sonst** -> Polygon: **3 Schritte: (1)** Berechnung der Schnittmenge zweier konvexer Polygone **(CPI = Convex Polygon Intersection)** -> ergibt konvexes Polygon, **(2)** Berechnung der Schnittmenge einer Menge von Halbebenen **(HPI = Half Plane Intersection)** -> konvertieren jeder Halbebene in konvexes Polygon und **(3) Normalisation.**

Implementierung von Algebra-Operationen für Constraint-Relationen

Mithilfe von **drei Hilfsoperationen:**
- **Normalize:** scannt und normalisiert symbolische Tupel
- **Satisfy:** prüft ob symbolische Relation erfüllbar ist
- **Overlap:** verarbeitet zwei symbolische Relationen und formt eine Konjunktion für jedes Paar symbolischer Tupel, wenn ihre bounding boxen sich überschneiden

(Ergebnis nicht in Normalform).

Implementierung:
- **Union** -> merge Tupel
- **Intersection | product | join | selection** -> durch Aufruf overlap
- **Projection** -> k-dim Fall: Fourier-Motzkin-Methode
- **Difference** -> Zelldekomposition: nur zur ersten Relation (und nicht zur zweiten) gehörende Zellen zurückgeben.

Spatio-Temporal Indexing (Part I)

Indexstrukturen

Hauptaufgabe: Schneller Zugriff auf Datensätze anhand eines Suchschlüssels (kein sequentieller Scan durch eine Datei). Bieten Operationen zur Unterstützung **spezieller Query-Typen**: exact match query | range query | sequential access

Indexstrukturen für **Standard (alphanumerische) Daten:**
- indexed sequencial method (**ISAM**)
- baumbasierte Indexstrukturen **(B | B+ | B*) - Bäume**
- hashbasierte Indexstrukturen **(external static | dynamic | extensible | linear) - Hashing**

Nichtstandard-Applikationen basieren auf komplexen Daten und Algorithmen und benötigen neue, anspruchsvollere Indexierungstechniken:
- **spatial data:** geometrische Indexstrukturen als multidimensionale spatial access methods **(SAM)**
- **temporal (standard) data:** temporale Indexstrukturen für valid/transaction-time (v.a. Diskrete Zeit)
- **spatio-temporal index Strukturen:** Indexierung stetiger Bewegung von spatial objects

Geometrische Grundlagen (rein geometrische Konzepte)

Indexierung Multidimensionaler Räume mit der R-Baum-Familie

R-Bäume sind höhenbalancierte Bäume und natürliche Erweiterungen des B-Baumes für k-Dimensionen. **Spatial Objects** werden in R-Bäumen durch ihre **MBBs** repräsentiert. Die **SAMs** der R-Baum-Familie sind datengesteuert.

Wie B-Baum -> Jeder Knoten (intern oder Blatt) auf eine Speicherseite abgebildet. **Anders als B-Baum** -> nicht auf Schlüssel mit totaler Ordnung definiert. MBBs anhand topologischer Enthaltenseinbeziehung organisiert. Jeder Knoten hat k-dimensionale Box,

die MBB aller MBBs seiner Kinder repräsentiert.

Eigenschaften R-Baum:
Knoten haben zwischen m und M **Einträge (MBB, nodeid|oid)** und enthalten räumlich MBBs seiner Kinder bzw das k-dimensionale Objekt. Alle **Blätter** auf gleicher Ebene. MBBs auf gleicher Ebene können sich überlappen (**keine exakte Partition** des Raumes).

Such-Operationen (wie Punkt- und Bereichsabfragen):
werden **top-down rekursiv** verarbeitet: prüfen ob **Punkt** in **MBB** eines Eintrages in **Wurzel** enthalten -> Suchalgorithmus rekursiv auf Einträge von **nodeid** anwenden, bis **Blatt** erreicht ist. Dann Einträge in **oid** verwenden um auf **Objekte** zuzugreifen. Bei **Region** Überlappung prüfen -> Suchpfad nicht eindeutig.
Suchperformance: abhängig von

- **coverage:** Gesamtfläche bedeckt von allen MBBs aller Knoten einer Ebene. Im Verhältnis zur Gesamtfläche aller indexierten Objekte -> Maß für leere bedeckte Fläche.
- **overlap:** von > 1 MBB bedeckte Fläche

Einfügen eines Objekts
Baum top-down rekursiv durchlaufen. Knoten wählen dessen MBB Objekt-MBB enthält oder durch sie minimal vergrößert wird. Auf Blattebene -> neuer Eintrag | Erweiterung + Propagation nach oben | split.

R* - Baum: Verbesserungen Einfüge-Algorithmus durch kombinierte gleichzeitige Optimierung von overlap, coverage und MBB-Umfang mit heuristischen Verfahren.
R+ - Baum: MBBs von Knoten auf einer Ebene überlappen nicht, für Punktanfrage gibt es nur einen einzigen eindeutigen Pfad Wurzel -> Blatt. Objekt-MBBs können auf verschiedene MBBs auf nicht-Blattebene verteilt werden -> R+-tree > R*-tree wg. dieser Duplikation (= clipping von MBBs). Effizientere Suchalgorithmen.

Duality

Mächtige Technik in geometrischen Algorithmen. Nutzt Tatsache, dass sowohl **Punkt** als auch **Linie** durch **2 Parameter** beschrieben werden können: **(x,y)-Koordinaten** bzw **slope + intercept**.
Duality Transformation: 1 zu 1 Abildung (primale -> duale Ebene) einer Menge von Punkten auf eine Menge von Linien und vice versa. **Abbildung** ist

- **incidence preserving** $p \in l \Leftrightarrow l^* \in p^*$
- **order preserving** p (above | below) l <=> l* (above | below) p*.
- Punkt p liegt in Streifen σ <=> Linie p* schneidet Liniensegment σ*

External Partition Tree

Populäre **Datenstruktur** für **geometrische Bereichsanfragen.** Basiert auf Konzept der **simplicial partition**. Menge in **disjunkte Untermengen** und alle Punkte einer Untermenge enthaltene **Dreiecke** unterteilt. Ein Punkt kann in 1 Untermenge und in vielen Dreiecken liegen.

- **Erzeugung:** rekursiv.

- **Suche:** top-down -> für jeden Knoten Dreiecke mit Querystrip vergleichen, auf Blattebene in Querystrip enthaltene Punkte zurückgeben.
- **Insertion und Deletion:** mittels partial rebuilding.

Catalog Structure

Für **Halbstreifenanfragen** (three-sided-range-queries) auf Punktmenge von B^2 Punkten. (maximal) 2B-1 **Blöcke** und konstante Zahl **Katalog-Blöcke**. Mit jedem Block **Rechteck** assoziiert -> Block enthält **Punkt**, wenn dieser innerhalb oder direkt oberhalb Rechteck liegt.

- **Konstruktion:** zunächst B Blöcke. Dann horizontaler sweep (Punkte zählend) -> neuer Block wenn zwei benachbarte Blöcke < B/2 Punkte oberhalb sweepline haben.
- **Halbstreifenanfrage** [a,b] x[c,∞]: Katalogblöcke in Hauptspeicher. Blöcke deren Rechteck Queryrechteck schneiden in Hauptspeicher laden und relevante Punkte zurückgeben.
- **Dynamisierung Struktur mit global rebuilding:** updates in Zusatzblock loggen -> wenn dieser voll ist komplette Struktur neu aufbauen.

External Priority Search Tree

Struktur benutzt einen **Basis-B-Baum** auf den x-**Koordinaten** der Punkte. Jeder **interne Knoten** speichert einen **x-Bereich,** der wiederum durch die **x-Bereiche** seiner **Söhne** in **slabs** unterteilt wird. Die obersten B Punkte jedes slabs werden in **Hilfs-Katalogstruktur** gespeichert.
Halbstreifenanfragen: Baum rekursiv top-down durchlaufen. Bei jedem Knoten wird Katalogstruktur befragt um nichtleere Schnittmengen mit Query zu finden.

External Range Tree

Ist eine **3-Ebenen-Struktu**r für **Halbstreifenanfragen:**

- **Primäre Struktur**: B-Baum über **x-Koordinaten** der Punkte. Söhne teilen x-Bereich in slabs.
- x-Koordinaten der **slab-Grenzen** in B-Baum gespeichert
- Die Punkte im x-Bereich eines Knoten in **4 Hilfsdatenstrukturen** gespeichert: **B-Baum** mit y-Koordinaten als Schlüssel sowie drei **externe priority search trees**.

Anforderungen an Indexierung von Moving Objects

Eigenheiten Spatio-Temporaler Indexstrukturen

Ziel: Spezifikations- und Klassifikationsschema für spatio-temporal access methods (STAM). **Eigenheiten** die Benutzung rein geometrischer Indexstrukturen (R-Tree, Quadtree) verhindern =>

- **MBBs** sich bewegender Objekte -> extrem große und überlappende Rechtecke mit viel dead space
- Monoton wachsende **Zeitwerte** können nicht einfach als 3.Dimension gesehen werden

-> ältere Instanzen können evtl ausgelagert oder eliminiert werden.
- Spezifische **Anfragearten** (timeslice|historic Queries) müssen unterstützt werden.

Spezifikationskriterien Spatio-Temporaler Indexstrukturen

- Unterstützte **Datentypen** und **Datenmengen: ST-Object:** (oid, s_i, b_i) => (Object_ID, spacestamp, timestamp). **Time-evolving Object:** als **Linie** (-> Punkt) oder **Volumen** (-> region) im 3D Raum. Es müssen **mpoints/mregions** mit **dynamischen Locations/Kardinalitäten** unter Berücksichtigung von **valid/transaction time** (=> chronological bzw append-only-data) sowie **bulk/dynamic loading** indexiert werden.
- **Indexkonstruktion: Insert/Split** -> rekursiv von Wurzel zu Blatt anhand räumlicher und zeitlicher Koordinaten. **Packen und Entfernen** obsoleter (Zeit-) Einträge. Index reorganisieren, wenn **timestamp-Granularität** geändert wird.
- **Query Processing: Selection** -> Schnittmenge Query-Window <-> MBB jedes Knoten in Raum und Zeit. **Join** -> paralleler Durchlauf von beiden Wurzeln: Knotenvergleich mit Overlap-Operator. **Nearest-neighbor-queries:** finde die k nächsten Objekte.

Überblick über STAM's in der Vergangenheit

- **Zeit als andere Dimension**: zB 3D R-Bäume. mObjects erzeugen viel dead space
- **Zeit als Zeitintervall in Datenstruktur** ohne neue Dimension (-> RT-Baum)
- **Überlappende Datenstrukturen** um Zustand DB zu verschiedenen Zeitpunkten (valid oder transaction) zu repräsentieren.

KE 7

Spatio-Temporal-Indexing (Part II)

Indexing Current and Near Future Movement

Allgemeine Strategien

In **konventionellen Indexstrukturen** werden statische und bis zum nächsten Update konstante Daten indexiert. Für **Daten** mit **stetiger Bewegung** -> entweder häufige Updates oder ungenaue Daten. **Lösung:** die Bewegung direkt als Fuktion der Zeit speichern.

Konzeptionelle Unterscheidungen:
- **indizierter Raum:** Trajectories als Linien im (d+1)-dimensionalen Raum | Punkte im höherdimensionalen Raum | Objekte im ursprüngliche d-dimensionalen Raum (Parametrisierung der Indexstruktur mit velocity Vektoren).
- **Partitionierung:** Daten (-> R-Baum) | Raum (-> Quadtree)

- **Ausmaß Datenreplikation:** verbessert Query-Performance, verschlechtert Update-Performance
- **Gültigkeit Index:** periodischer Neuaufbau | unendlich gültig (gelegentliche Optimierung um „Index-Fäulnis“ vorzubeugen)

Time-parameterized R-Tree (TPR-Tree)

Natürliche Erweiterung des R*-Baumes. Benutzt eine **lineare Funktion** pro indexierten mpoint mit Parametern **position** und **velocity vector.** Die **bounding boxes** des Baumes sind auch **Funktionen der Zeit** -> folgen dem mpoint kontinuierlich. Kann wie R-Baum Punkte im 1D, 2D oder 3D-Raum indexieren. Indexiert im native space, ohne Replikation und ohne periodischen Index-Neuaufbau. Ermöglicht Vorhersage für die (nahe) Zukunft und löst Problem der häufigen Updates.
Parameter (**position** und **velocity vector)** werden auch benutzt um Bounding Boxen als Funktion der Zeit darzustellen. Zuweisung Objekt zu Bounding Box sollte berücksichtigen, wann die meisten Anfragen eintreffen.

Query-Typen: (timeslice | window | moving) Queries spezifizieren Hyper-Rectangle/ Trapez in Zeitpunkt/Intervall. **W** ist Verallgemeinerung von **T** und Spezialisierung von **M**. Query Q hängt von Zeitpunkt der Anfragestellung **iss(Q)** ab.
- **T:** wie R-Baum, aber BBs müssen für iss(Q) berechnet werden bevor Überschneidung mit Query-region geprüft wird.
- **W, M:** Prüfen ob innerhalb Query-Zeitintervall Query-Trapez das TPBB-Trapez schneidet. Für 1D -> einfach, für (≥) 2D -> allgemeiner Polyeder-Polyeder-Intersection Test

Problemparameter TPR-Baum:
Time Horizon **H** | Index Usage Time **U** | Querying Window **W. H = U + W.**

Struktur des TPR-Baumes: Balancierter Vielwegbaum mit Struktur R-Baum.
- **Interne Knoten:** Zeiger auf Unterbaum und auf Rechteck das alle mpoint-Positionen/MBBs enthält.
- **Blattknoten:** Position mpoint und Zeiger auf Objekt in DB.
- **Konservative Bounding Boxes (CBB):** zu einem Zeitpunkt minimal, später nicht mehr. Obere/untere Grenze bewegen sich mit maximaler/minimaler Geschwindigkeit -> CBB schrupft niemals. Die **load-time TPBB** (time parameterized bounding box) wird bei updates durch die meist kleinere **update-TPBB** ersetzt.

Heuristiken für Baumorganisation: Wie gruppiert man mObjects so in Knoten, das timeslice-queries mit Zeithoriziont H möglichst effizient unterstützt werden?
- **H -> 0:** existierende R-Baum Algorithmen für Einfügen und Bulkloading.
- **H ist groß:** Gruppierung mpoints entsprechend velocity vectors -> kleine BB-Wachstumsraten. Optimierung der Integrale der Zielfunktionen (anstatt der ZF selber wie beim R*-Baum)

Dual Data Transformation Approach

Dualität bildet eine Hyperebene in R^d auf einen Punkt in R^d ab -> intuitivere Problemformulierung und einfachere Lösung.
Trajectories von **mpoints** als **Linien** in der **time-location (t,y) Ebene** dargestellt. In der **dualen Ebene** stellt eine Achse dann den **slope**, die andere den **intercept** der Trajectory dar. Eine (Trajectory-) Linie in der primalen Ebene wird zum **Punkt** in der dualen Ebene (**Hough-X-Transform**).
Auch eine **window Query W** muß in die duale Ebene transformiert werden (-> mittels linearer constraints als Polygon ausgedrückt). **Zwei Indexstrukturen** benutzen um monoton wachsende, unbegrenzte **Zeitachse** zu kontrollieren.

Problem der **Indexierung von mobjects** auf einer **Linie** wird auf **einfache Bereichssuchen in 2D** transformiert -> finde alle Punkte innerhalb des Querypolygons. Dieses Problem wurde bereits intensiv in **statischer Hauptspeicher Umgebung** untersucht. Beim **dynamischen Sekundärspeicher-Setting** übermittelt ein Speicherzugriff aber B (=Seitenkapazität) Dateneinheiten in einer einzelnen Operation.

Eine beinahe optimale Lösung für einfache Bereichssuchen basiert auf **partition trees** und **simplicial partitions** (und **dynamisierten Strukturen** beim Externspeicher-Algorithmis). Aber: **partition trees** sind in der **Praxis nicht** sehr **nützlich**, da ein versteckter konstanter Faktor sehr groß werden kann.
Zwei Ansätze für Verbesserung der **average query time**:

- Vorhandene **Zugriffsmethoden** für **orthogonale Bereichsanfragen** an **nicht-orthogonale-queries (wie simplex-range-queries)** anpassen. Für Bereichsanfragen im dualen Raum (mit sehr schiefer Punktverteilung) scheinen **kd-Bäume** besser geeignet zu sein als **R-Bäume**. Letztere verzweigen nur anhand einer Dimension (intercept) und clustern Datenpunkte in annähernd quadratische Regionen, während kd-Bäume anhand beider Dimensionen verzweigen und performanter sind.
- Eine **Linie** anstatt als **y(t) = vt + a** durch **t = (1/v) y – (a/v)** repräsentieren. Die Abbildung dieser Linienrepräsentation in die duale Ebene **(Hough-Y-Transform)** erzeugt einen Punkt mit Koordinaten **(1/v)** und **– (a/v).** Eine Anfrageregion wird als Schnittmenge zweier Halbebenen-Anfragen gegeben.

Indexieren in zwei Dimensionen

Spezialfall (1.5 D):
mobjects bewegen sich auf **Netzwerk** in der Ebene. Durch Darstellung jeder **Route** im Netzwerk als Sequenz verbundener gerader Segmente **(polyline)** kann Problem auf eine Anzahl von **1D-Queries** reduziert werden. Indexierung mit Standard-SAMs, window query bestimmt Intersection Route <-> Query-Rechteck und gibt Menge (zT unverbundener) Liniensegmente zurück.

Allgemeiner Fall (2 D):
freie Objektbewegung in beschränkter Fläche der Ebene. **Trajectories** von mobjects als Linien in 3D, **window queries** als cube (x,y,t) in 3D dargestellt. **Queryergebnis** -> Menge von 3D-Linien die Query-Cube schneiden.
Dualer Transformationsansatz kann nicht direkt von Linien im **(y,t)-Raum** auf Linien

im **(x,y,t)-Raum** übertragen werden, da Linien im 3D-Raum **4 Freiheitsgrade** haben: die Entwicklung jeder **Koordinate** x und y kann unabhängig als lineare Funktion der Zeit mit je 2 Parametern beschrieben werden
=> **3D-Linien** werden auf **(x,t)** und **(y,t)-Ebene** projeziert und nach **2 dualen Transformationen** als **4D-Punkt** repräsentiert, **query-window** wird nach Projektion zu **wedge (=Keil)** wie in 1D. **Schnittmenge** der Antworten beider Projektionen -> Antwort Originalanfrage.
Alternativ: 4D-Partition-Tree | Index auf Basis kd-Baum

Time-oblivious Indexing with Multilevel-Partition-Trees

Wenn bei dualer Transformation jede Linie in 3D auf Punkt in R^4 abgebildet wird und 4 Partition-Trees für Bereichsanfragen genutzt werden ist Query-Zeit recht groß.
Verbesserung durch **Multi-Level-Partition-Tree** -> **komplexe Anfragen** beantworten, indem man sie in mehere **einfachere Komponenten zerlegt** und jeder Komponente eine **separate Datenstruktur** zuweist.
Duale Transformation: jeder mpoint in (x,y)-Ebene auf zwei statische Punkte *p* in (x,t) und (y,t) - Ebene abgebildet, jedes Query-Rechteck auf 2 Querystreifen *w*.
Queryergebnis sind alle Punkte für die gilt: $p^x \in w^x \wedge p^y \in w^y$
Konstruktion: **Primärer Partition Tree** auf x-Projektion, an bestimmten Knoten **sekundärer Partition** Tree auf y-Projektion.
Bereichsanfrage: zuerst im primäry partition tree suchen. Für Dreiecke die komplett in Querystreifen liegen, keine komplette depth-first-Suche, sondern sekundären partition tree benutzen.

Kinetic B-Trees

Datenstruktur basiert auf **kombinatorischen Eigenschaften** wie der sortierten Ordnung der Punkte entlang der x- und y-Achse. **Kinetische Events** (Ordnung der Punkte ändert sich) lösen **kinetische Updates** aus.
Da Bewegung der Punkte bekannt ist, können events vorhergesagt werden -> **globale event queue (**=priority queue). Wenn ein swap von 2 benachbarten Punkten erfolgt, werden neue events in queue eingefügt und nicht mehr relevante events entfernt.
Dadurch wird zugrundeliegender **B-Tree** periodisch upgedatet.
Queries nur für die Gegenwart (oder in chronologischer Reihenfolge für die Zukunft).

Kinetic External Range Trees

basiert auf external range trees, der für Bereichsanfragen auf mpoints kinetisiert werden muß. Modifikationen bottom-up:

- **Catalog structure:** Koordinaten der Rechtecke der Blöcke durch Koordinatenfunktionen der Zeit ersetzen, damit Rechtecke sich stetig mit der Zeit verändern können.
- **External priority search tree/ External range tree:** Veränderungen bei kinetischen events (swaps)

Time-Responsive Indexing with Multiversion Kinetic B-Trees

- **partition tree** -> beliebige Abfolge von time-range Queries mit hohen Kosten

- **kinetic range trees** -> nur chronologische Reihenfolge, aber schnell
- **multiversion kinetic B-Trees** -> Kombination: schnelle Antwort auf Queries in beliebiger Reihenfolge. Man speichert past und future version des Baumes (nur die Veränderungen). Elementen und Knoten werden Lebensspannen zugeordnet.

Time-Responsive Indexing with Multiversion External Kinetic Range Trees

3 Ebenen: 1. und **2.** spezielle **B-Bäume**, **3. Katalogstrukturen** an jedem Knoten eines priority search trees => verwalten **Updateblock** für die Aufzeichnung **kinetischer Events**. Nach B events wird Katalogstruktur neu aufgebaut. Mehere Versionen der Katalogstrukturen und des priority search trees gespeichert.

Indexing Trajectories (History of Movement)

Kontepte

Zeitliche Entwicklung räumlicher Objekte in der Vergangenheit, insbesondere Querying und Indexieren von mpoint Objekten. Besonderheit: ST-data ist append-only in Bezug auf Zeitdimension und sie wachsen vor allem in der Zeitdimension. Zwei geeignete Zugriffsmethoden ->

Spatio-Temporal-R-Tree (STR-Tree)

Organisiert Liniensegmente nicht nur nach **räumlichen Eigenschaften** sondern gruppiert sie auch nach **Trajectory**. **Neues Format** der **Einträge** in den **Blättern**: (mbb, oid, traj_id, orientation) mit orientation ϵ {1,2,3,4}.
STR-Baum erweitert den R-Baum (-> viel dead space und damit viel Überlappung und geringe Diskriminationsfähigkeit): verändertes Blattformat (s.o.) und modifizierte Enfüge/Splitstrategie:

Einfügen:
- **R-Baum:** rein räumliche Einfüge-Prozedur basierend auf räumlicher Nähe.
- **STR-Baum:** räumliche Nähe und teilweise Trajectory-Erhaltung. **Ziel:** Segmente einer Trajectory zusammenhalten und neue Liniensegmente so nahe wie möglich bei ihrem Vorgänger in der Trajectory einfügen.

Splitting Strategie:
- **Blattknoten:** (**1**) alle Segmente disconnected -> Standard R-Split-Algorithmus, **(2)** nicht alle, aber mindestens 1 Segment disconnected -> disconnected Segments in neu erzeugten Knoten, **(3)** kein Segment disconnected -> zeitlich jüngstes backward-connected Segment in neu erzeugten Knoten.
- **Nicht-Blattknoten:** neuen Eintrag in neuen Knoten einfügen und alten Knoten behalten wie er ist.

Trajectory-Bundle Tree (TB-Tree)

konzentriert sich ganz auf Trajectory-Erhaltung und läßt andere spatial properties außer

acht.

- **STR-Baum** ist R-Baum basiert und erreicht Trajectory-Orientierung über neue Einfüge und split-Atrategie, ohne die räumliche Diskriminierungsfähigkeiten des Indexes stark zu beeinflußen. **Zentrale Annahme R-Baum:** alle eingefügten Geometrien (zB Liniensegmente) sind unabhängig. **Wissen über Trajectories** ist **implizit** vorhanden.
- **TB-baum:** Strikte Erhaltung von Trajectories indem Blattknoten nur Liniensegmente aufnehmen die zur selben Trajectory gehören: **Index als Trajectory-Bundle.** Räumliche Diskriminierung des R-Baumes wird aufgegeben, dafür besser bei rein ST-Queries. **Nachteil:** benachbarte Segmente in verschiedenen Knoten gespeichert, wenn sie nicht zur selben Trajectory gehören.

Blattknotenformat: (mbb, oid, orientation), traj_id überflüssig.

- **Einfügen:** Mit Algorithmus FindNode Blatt suchen, indem Vorgäger gespeichert. Wenn voll ->
- **Split:** einfügen in neues Blatt im am weitesten rechts gelegenen Pfad -> TB-Baum wächst von links nach rechts.
- **Struktur TB-Baum:** Menge von Blattknoten mit partiellen Trajectories organisiert in Baumhierarchie. Komplette Trajectory über über unverbundene Blattknoten verteilt. Verbindung mit doppelt verlinkter Liste um Query anhand Trajectory-ID zu ermöglichen -> erhält Trajectory-Evolution.

Query Processing

3 Haupttypen von Anfragen:
Coordinate based queries: point, range (window), nearest neighbor queries in 3D-space. Wie klassische Bereichsanfragenverarbeitung mit R-Baum: Baum top-down mit Koordinaten -Constraints durchlaufen bis Blatt.
Trajectors based queries: topological (ST-Prädikaten Disjoint, Meet, Overlap... -> zu gewöhnlichen range-queries reduziert), **navigational** (speed, area... -> leiten Information aus Trajectories ab)
Combined queries: verbinden coordinate-based und topological-queries (andere Algorithmen). Alle entries in **range** <u>und</u> entries der selben **trajectory**. **(1) R- und STR-Baum:** erst range-search, dann partielle Trajectories extrahieren. **(2) TB-Baum:** Verlinkte Listen erlauben es verbundene Segmente ohne ohne Suchen zu erhalten, im selben oder in anderen Blättern.